UNITED STATES PATENT APPLICATION

FOR

METHOD AND APPARATUS FOR DETECTING CORRUPT SOFTWARE CODE

INVENTORS:

Gunawan Ali-Santosa, a citizen of the United States
Mehrdad Mojgani, a citizen of Iran

ASSIGNED TO:

Sun Microsystems, Inc., a Delaware Corporation

PREPARED BY:

THELEN, REID & PRIEST LLP
P.O. BOX 640640
SAN JOSE, CA 95164-0640
TELEPHONE: (408) 292-5800
FAX: (408) 287-8040

Attorney Docket Number: SUN-P7207

Client Docket Number: SUN-P7207

METHOD AND APPARATUS FOR DETECTING CORRUPT SOFTWARE CODE

Cross Reference to Related Applications

[0001] This application is related to the following:

U.S. patent application filed November 13, 2001 in the name of inventors Gunawan Ali-Santosa and Mehrdad Mojgani, entitled "Method and Apparatus for Remote Software Code Update", Attorney Docket No. SUN-P6542, commonly assigned herewith.

U.S. patent application filed November 13, 2001 in the name of inventors Gunawan Ali-Santosa and Mehrdad Mojgani, entitled "Method and Apparatus for Managing Remote Software Code Update", Attorney Docket No. SUN-P7206, commonly assigned herewith.

FIELD OF THE INVENTION

[0002] The present invention relates to the field of computer science. More particularly, the present invention relates to a method and apparatus for updating and detecting corruption in software code downloaded to a remote device.

BACKGROUND OF THE INVENTION

[0003] The software code executing in remote or embedded devices often needs to be updated both during development of the embedded device and after the device has been delivered to the customer (post-issuance). Typically, several code updates are required during product development. The reliability and efficiency of updates during

development of the embedded device affect the time required to develop the product. The reliability and efficiency of post-issuance updates is also important, because non-functioning devices typically must be returned to the device issuer, resulting in shipping-related costs and delays.

[0004] One type of update is performed to fix "bugs" or problems with the software. Another type of update is performed to include new product features, such as special functions tailored for particular customers. A code update may fail for a number of reasons, including operator error, power failure and other unexpected events. These failure events sometimes make the device inoperable. These failure events may also make the device appear to function normally despite the existence of a failure within the code space. This latent problem may manifest itself subsequently during normal operation, often resulting in an inoperable device. Therefore, it is important to detect such a failure prior to program execution.

[0005] Such failure conditions are typically detected by reading back each byte of code previously written and comparing it against an expected value. This process is explained in more detail with reference to FIG. 1.

[0006] Turning to FIG. 1, a flow diagram that illustrates a typical method for detecting corrupted code by comparing the value of each storage unit downloaded to device with the corresponding value of the storage units read back from the device is presented. At 100, the code is downloaded to the embedded device. At 105, a previously

downloaded byte is read from the embedded device. At 110, the byte is compared to the expected value. At 115, a determination is made regarding whether the byte value matches the expected value. At 120 an indication that the code is corrupted is made if the byte value does not match the expected value and the process ends at 135. If the byte value matches the expected value, at 125 a determination is made regarding whether there is another byte to check. This process continues until all bytes in the program have been checked. If all of the bytes have been checked and all of them match their expected values, an indication that the program is valid is made at 130.

[0007] Unfortunately, this method of reading back every byte and comparing it to an expected value is time-consuming. Additionally, the expected values for particular bytes may change with each software version, thus requiring special knowledge about each software version.

[0008] An improvement is made possible by using a checksum. This process of using a checksum is explained in more detail with reference to FIG. 2.

[0009] Turning now to FIG. 2, a flow diagram that illustrates a typical method for using a checksum to detect corrupted code is presented. At 200, the code is downloaded to the embedded device. At 205, a checksum is initialized. At 210, a byte previously downloaded is read. At 215, the byte value is added to the checksum. At 220, a determination is made regarding whether there is another byte to check. This process continues until the value of each program byte has been added to the checksum. At 225,

the calculated checksum is compared to the expected checksum. If the calculated checksum and the expected checksum are not the same, an indication that the code is corrupted is made at 230 and the process ends at 240. If the calculated checksum and the expected checksum match, an indication that the code is valid is made at 235.

[0010] Unfortunately, this checksum method requires adding each byte, reading it back and comparing it against an expected value. While this method typically takes less time than reading and comparing every byte to an expected value, the method is still time-consuming. Additionally, the checksum method sometimes fails to detect corrupt code. In these cases, the calculated checksum of corrupted code matches the expected checksum, causing corrupt code to be used.

[0011] Once the determination that the code is corrupted is made, other code that is not corrupt must be executed. One typical solution is to provide complete redundancy by maintaining two copies of the code. One copy is typically maintained in "boot" flash, while the other is maintained in "main" flash. This is illustrated by FIG. 3.

[0012] Turning now to FIG. 3, a flow diagram that illustrates a method for updating code on an embedded device using separate copies of the program in boot flash memory and main flash memory is presented. At 300, the code is downloaded. At 305, a first copy of the program code is maintained in main flash memory and a second copy of the program code is maintained in boot flash memory. At 310, execution begins from boot flash memory. At 315, a determination is made regarding whether main flash memory is

corrupted. If main flash memory is corrupted, the copy of the program code in boot flash memory is executed at 320. If main flash memory is not corrupted, a copy of the program code in main flash memory is executed at 325.

[0013] This solution provides complete redundancy so that one copy of the code may be executed when the other is corrupted. However, this redundancy also requires twice the memory, thus constraining the maximum program size in embedded devices that follow this approach.

[0014] Other solutions provide additional levels of redundancy. But this additional redundancy typically comes at the expense of even higher memory requirements, further constraining the maximum program size.

SUMMARY THE INVENTION

[0015] A method for detecting corrupt software code includes defining a correlation rule for a program that includes at least one segment that includes multiple markers. The correlation rule defines a relationship between two or more of the markers. The method also includes writing the program to a memory device, reading two or more of the markers from the memory device, determining whether a segment is corrupt by applying the correlation rule to the two or more markers and indicating whether the segment is corrupt based upon the determining. According to one aspect, the correlation rule includes comparing the content of a first memory location with the content of a second memory location. According to another aspect, the first memory location is the first memory location of a segment and the second memory location is the last memory location of the segment. According to another aspect, the first memory location is a memory location of a first segment and the second memory location is a memory location of a second segment. According to another aspect, the indicating includes indicating a segment is corrupt when the content of a first memory location does not equal the complement of the content of the second memory location.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] The accompanying drawings, which are incorporated into and constitute a part of this specification, illustrate one or more embodiments of the present invention and, together with the detailed description, serve to explain the principles and implementations of the invention.

[0017] In the drawings:

FIG. 1 is a flow diagram that illustrates a typical method for detecting corrupted code by comparing the value of each storage unit downloaded to device with the corresponding value of the storage units read back from the device.

FIG. 2 is a flow diagram that illustrates a typical method for using a checksum to detect corrupted code.

FIG. 3 is a flow diagram that illustrates a method for updating code on an embedded device using separate copies of the program in boot flash memory and main flash memory.

FIG. 4 is a block diagram that illustrates a computer system suitable for implementing aspects of the present invention.

FIG. 5 is a block diagram that illustrates a method for segmenting memory in accordance with one embodiment of the present invention.

FIG. 6A is a block diagram that illustrates configuring a page register in accordance with one embodiment of the present invention.

FIG. 6B is a block diagram that illustrates configuring a virtual memory register in accordance with one embodiment of the present invention.

FIG. 7A is a code sample that illustrates mapping main flash memory in accordance with embodiments of the present invention.

FIG. 7B is a code sample that illustrates mapping boot flash memory in accordance with embodiments of the present invention.

FIG. 8 is a flow diagram that illustrates mapping flash memory in accordance with embodiments of the present invention.

FIG. 9 is a flow diagram that illustrates a method for updating code in accordance with one embodiment of the present invention.

FIG. 10 is a flow diagram that illustrates a method for detecting corrupted code in accordance with one embodiment of the present invention.

FIG. 11 is a block diagram that illustrates the state of main flash memory and boot flash memory during power-up in accordance with one embodiment of the present invention.

FIG. 12 is a block diagram that illustrates the state of main flash memory and boot flash memory when the code in main flash memory is preparing to transfer program control to boot code in accordance with one embodiment of the present invention.

FIG. 13 is a block diagram that illustrates the state of main flash memory and boot flash memory when passing program control to boot flash memory in accordance with one embodiment of the present invention.

FIG. 14 is a block diagram that illustrates the state of main flash memory and boot flash memory after passing program control to boot flash memory in accordance with one embodiment of the present invention.

FIG. 15 is a block diagram that illustrates the state of main flash memory and boot flash memory when mapping a main flash code segment to data space in accordance with one embodiment of the present invention.

FIG. 16 is a block diagram that illustrates the state of main flash memory and boot flash memory when a corrupt boot flash memory segment is detected in accordance with one embodiment of the present invention.

FIG. 17 is a block diagram that illustrates the state of main flash memory and boot flash memory after program control is transferred to main flash memory upon detecting a corrupt boot flash memory segment in accordance with one embodiment of the present invention.

FIG. 18 is a block diagram that illustrates the state of main flash memory and boot flash memory when a corrupt main flash memory segment is detected in accordance with one embodiment of the present invention.

FIG. 19 is a block diagram that illustrates the placement of markers in main flash memory in accordance with one embodiment of the present invention.

FIG. 20 a block diagram that illustrates the placement of markers in boot flash memory in accordance with one embodiment of the present invention.

FIG. 21A is a table that illustrates correlation between markers in a main flash memory in accordance with one embodiment of the present invention.

FIG. 21B is a table that illustrates correlation between markers in a boot flash memory in accordance with one embodiment of the present invention.

FIG. 22 is a flow diagram that illustrates a method for detecting corrupted code in accordance with one embodiment of the present invention.

FIG. 23 is a flow diagram that illustrates a method for determining whether main flash memory has been corrupted in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION

[0018] Embodiments of the present invention are described herein in the context of a method and apparatus for updating and detecting corruption in software code downloaded to a remote device. Those of ordinary skill in the art will realize that the following detailed description of the present invention is illustrative only and is not intended to be in any way limiting. Other embodiments of the present invention will readily suggest themselves to such skilled persons having the benefit of this disclosure. Reference will now be made in detail to implementations of the present invention as illustrated in the accompanying drawings. The same reference indicators will be used throughout the drawings and the following detailed description to refer to the same or like parts.

[0019] In the interest of clarity, not all of the routine features of the implementations described herein are shown and described. It will, of course, be appreciated that in the development of any such actual implementation, numerous implementation-specific decisions must be made in order to achieve the developer's specific goals, such as compliance with application- and business-related constraints, and that these specific goals will vary from one implementation to another and from one developer to another. Moreover, it will be appreciated that such a development effort might be complex and time-consuming, but would nevertheless be a routine undertaking of engineering for those of ordinary skill in the art having the benefit of this disclosure.

[0020] In the context of the present invention, the term "network" includes local area networks, wide area networks, the Internet, cable television systems, telephone systems, wireless telecommunications systems, fiber optic networks, ATM networks, frame relay networks, satellite communications systems, and the like. Such networks are well known in the art and consequently are not further described here.

[0021] In accordance with one embodiment of the present invention, the components, processes and/or data structures may be implemented using programs running on computers. Different implementations may be used and may include other types of operating systems, computing platforms, computer programs, firmware, computer languages and/or general-purpose machines. In addition, those of ordinary skill in the art will readily recognize that devices of a less general purpose nature, such as hardwired devices, devices relying on FPGA (field programmable gate array) or ASIC (Application Specific Integrated Circuit) technology, or the like, may also be used without departing from the scope and spirit of the inventive concepts disclosed herein.

[0022] According to embodiments of the present invention, updating code for a first program resident in a first code space includes transferring program control to a second program executing in a second code space, mapping a program segment to data space and writing the segment. According to other embodiments of the present invention, detecting corrupt software code includes defining a correlation rule between two or more markers in a program, downloading the program to an embedded device and using the correlation rule to determine whether the downloaded program is corrupt.

[0023] Figure 4 depicts a block diagram of a computer system 400 suitable for implementing aspects of the present invention. As shown in FIG. 4, computer system 400 includes a central processing unit (CPU) 415 coupled to a workstation 410. The CPU 415 is also coupled to an embedded device 420. The embedded device 420 includes a processor 425 and a memory 430

[0024] According to one embodiment of the present invention, CPU 415 comprises a SPARC IIe CPU, available from Sun Microsystems, Inc., of Palo Alto, CA.

[0025] According to another embodiment of the present invention, processor 425 comprises a DS80CH11 System Energy Manager, available from Dallas Semiconductor Corp. of Dallas, TX and memory 530 comprises a Wafer Scale WS 833, available from ST Microelectronics, of San Jose, CA.

[0026] Many other devices or subsystems (not shown) may be connected in a similar manner. Also, it is not necessary for all of the devices shown in FIG. 4 to be present to practice the present invention, as discussed below. Furthermore, the devices and subsystems may be interconnected in different ways from that shown in FIG. 4. The operation of a computer system such as that shown in FIG. 4 is readily known in the art and is not discussed in detail in this application, so as not to overcomplicate the present discussion. Code to implement the present invention may be operably disposed in memory 430 or stored on storage media such as a fixed disk, floppy disk or CD-ROM.

[0027] In operation, user 435 enters commands via terminal 410. Upon receiving a code update command, CPU 415 sends a code update command to embedded processor 425. Embedded processor 425 receives the code update command, transfers program control to boot code segment 0, selects a code segment to update, remaps that code segment to data space, erases the segment in data space and then writes the segment. This process continues for all code segments. Once the required number of code segments have been updated, program control is transferred to boot flash memory segment 0, the embedded device is reset and execution begins from boot code segment 0. A code integrity check is performed and if all segments are satisfactory, program control is transferred to main flash memory segment 0 and normal operation resumes. If at least one segment is unsatisfactory, program control is transferred to boot code, and the user is given the opportunity to retry the operation.

[0028] According to one embodiment of the present invention, the code integrity check compares bytes in predetermined locations and compares each byte to a corresponding expected value. This is described in more detail below with reference to FIGS. 18-23.

[0029] Turning now to Figure 5, a block diagram that illustrates segmenting memory in accordance with one embodiment of the present invention is presented. Figure 5 illustrates segmenting code space 500 and data space 502. The code space is divided into four 16K main flash memory segments and four 8K boot flash memory segments. Main

flash code segments 0 (522), 1 (520), 2 (518) and 3 (516) are mapped to address ranges 0000-3FFF, 4000-7FFF, 8000-BFFF and C000-FFFF, respectively. Boot flash code segments 0 (514), 1 (512), 2 (510) and 3 (508) are mapped to address ranges 0000-1FFF, 2000-3FFF, 4000-5FFF and 6000-7FFF, respectively, in the code space.

[0030] The data space is also divided into four 16K main flash memory segments and four 8K boot flash memory segments. However, all four main flash memory segments 534 are mapped to the same address range (4000-7FFF), and all four boot flash memory segments 532 are mapped to the same address range (4000-5FFF).

[0031] The memory mapping architecture presented in FIG. 5 is for illustrative purposes only. Those of ordinary skill in the art will recognize that the invention may apply to a different number of segments as well as to segments having various names, sizes and address ranges.

[0032] According to one embodiment of the present invention, boot segment 0 is configured with code that allows other segments to be updated and application-specific code is placed in one or more other segments. Configuring boot segment 0 this way decreases the probability that boot segment 0 will require an update, thus increasing reliability.

[0033] Figures 6A-6B are block diagrams that illustrate methods for segmenting memory in accordance with embodiments of the present invention. Figures 6A and 6B

represent registers available for configuring address mapping in a WS 833, available from ST Microelectronics of San Jose, CA. Figure 6A illustrates configuring a page register. Figure 6B illustrates configuring a virtual memory register. Figures 6A-6B are for illustrative purposes only. Those of ordinary skill in the art will recognize that other memory devices with various control registers may be used.

[0034] Referring to FIG. 6A, field 605 is used to select a main flash memory segment. Field 610 is used to select a boot flash memory segment. Field 615 is used to select whether main flash or boot flash code is executed.

[0035] Referring to FIG. 6B, field 635 indicates whether code is executed out of boot flash memory. Field 640 indicates whether code is executed out of main flash memory. Field 645 indicates whether data is boot flash data. Field 650 indicates whether data is main flash data.

[0036] Figures 7A-7B are code samples that illustrate mapping memory in accordance with embodiments of the present invention. Figure 7A illustrates mapping main flash memory. Figure 7B illustrates mapping boot flash memory.

[0037] Referring to FIG. 7A, note that when main flash is code, the segment number is determined by the address range, and when the device space is data, the address range is the same for all segments.

[0038] Referring to FIG. 7B, note that when boot flash is code, the segment number is determined by the address range, and when the device space is data, the address range is the same for all segments.

[0039] Turning now to FIG. 8, a flow diagram that illustrates mapping memory in accordance with embodiments of the present invention. Figure 8 uses a flow chart to illustrate the memory mapping shown in FIGS. 7A and 7B. At 800, a determination is made regarding whether the current device space is code or data. If the current device space is code, at 802 a determination is made regarding whether the current address is in the range defined for main flash memory. If the current address is in the range defined for main flash memory, determinations are made at 804, 806, 808 and 810 regarding whether the currently selected main flash memory segment is main flash segment 0, 1, 2 or 3, respectively. A positive determination results in a corresponding main flash segment indication at reference numerals 822, 824, 826 and 828. If the results of determinations 804, 806, 808 and 810 are negative, a determination is made at 812 regarding whether the current address is in the range defined for boot flash memory. If the current address is in the range defined for boot flash memory, determinations are made at 814, 866, 818 and 820 regarding whether the currently selected boot flash memory segment is 0, 1, 2 or 3, respectively. A positive determination results in a corresponding boot flash segment indication at reference numerals 830, 832, 834 and 836.

[0040] Still referring to FIG. 8, if at 800 it is determined that the current device space is data, a determinations is made regarding whether the current flash memory segment is a main flash memory segment. If the current flash memory segment is a main flash memory segment determinations are made at 840, 842, 844 and 846 regarding whether the current address is in the range defined for main flash segment 0, 1, 2 or 3, respectively. A positive determination results in a corresponding main flash segment indication at reference numerals 822, 824, 826 and 828. If at 838 the current flash memory segment is a boot flash memory segment, determinations are made at 848, 850, 852 and 854 regarding whether the current address is in the range defined for boot flash memory segments 0, 1, 2 and 3, respectively. A positive determination results in a corresponding boot flash segment indication at reference numerals 830, 832, 834 and 836.

[0041] Figures 9 and 10 are flow diagrams that illustrate methods for updating code and detecting corrupt code in accordance with embodiments of the present invention. Figure 9 illustrates updating code from the perspective the CPU (reference numeral 415 of FIG. 4). Figure 10 illustrates detecting corrupted code from the perspective of the embedded device (reference numeral 420 of FIG. 4).

[0042] Turning now to FIG. 9, a flow diagram that illustrates a method for updating code in accordance with one embodiment of the present invention is presented. Figure 9 is shown from the perspective of the CPU (reference numeral 415 of FIG. 4). At 900, an update command is received. At 905, program control is transferred to boot code

segment 0. At 910, a segment to write is selected. At 915, the selected segment is remapped to data space. At 920, the segment is erased. At 925, the segment is written. At 930, a determination is made regarding whether all segments have been updated. If another segment needs to be updated, the segment is updated beginning at reference numeral 910. If all segments have been updated, at 940 program control is transferred to segment 0. At 945 the embedded device is reset. At 950, program execution proceeds from boot code segment 0. At 955, code integrity is checked. At 960, a determination is made regarding whether any segment is corrupt. If at least one segment is corrupt, at 965 execution proceeds from a boot flash memory segment and at 970 the user is given the opportunity to retry the update. The update process begins at 900. If no segments are corrupt, execution proceeds from main flash memory segment 0 and normal operation is resumed at 980.

[0043] Turning now to FIG. 10, a flow diagram that illustrates a method for detecting corrupted code in accordance with one embodiment of the present invention is presented. Figure 10 is shown from the perspective of the embedded device (reference numeral 420 of FIG. 4). At 1000, the embedded device executes from boot code. At 1005, a self-check of main flash is performed. At 1010, the results of the self-check are examined. If the self-check results indicate main flash is invalid, an indication to that effect is made at 1015, an indication that the operator must restore main flash is made at 1055 and the CPU enters a "protected" mode, filtering subsequent communications to the embedded device at 1020. The communications are filtered to prevent normal commands from being sent to the embedded device before main flash memory is successfully updated

with valid code. Such command filtering prevents the embedded device from entering an unpredictable state.

[0044] According to one embodiment of the present invention, the commands allowed to be sent to the embedded device while in protected mode comprises commands to select boot flash memory or main flash memory, commands to select a flash memory segment, commands to erase flash memory and commands to write flash memory.

[0045] Still referring to FIG. 10, if the self-check results indicate main flash is valid, an indication to that effect is made at 1025 and the embedded device executes from main flash at 1030. At 1035, a self-check of boot flash is performed. At 1040, the results of the self-check are examined. If the self-check results indicate boot flash is invalid, an indication that the operator must restore boot flash is made at 1050. If the self-check results indicate boot flash is valid, normal operation of the embedded device is resumed at 1045.

[0046] Still referring to FIG. 10, if the embedded device is executing from main flash memory, an indication that main flash memory is valid is made at 1025. At 1030, a self-test result is received from the embedded device. At 1035, a determination is made regarding whether the self-test was successful. If the self-test was successful, normal operation is resumed at 1040. If the self-test was unsuccessful, at 1045 an indication that the operator must restore boot flash memory is made. Execution continues at 1010 if the self-test was unsuccessful.

[0047] According to one embodiment of the present invention, the program in main flash memory is a main program and the program in boot flash memory is a boot program. The main program includes full program functionality.

[0048] According to another embodiment of the present invention, the code space associated with the main program is larger than the code space associated with the boot program. Here, a main program that is larger than the code space associated with the boot program cannot be duplicated in the boot program code space.

[0049] According to another embodiment of the present invention, the main program is larger than the boot program.

[0050] Figures 11-15 the state of code memory for a process that begins with power-up (FIG. 11) and ends with updating main flash code (FIG. 15).

[0051] Turning now to FIG. 11, a block diagram that illustrates the state of main flash memory and boot flash memory during power-up in accordance with one embodiment of the present invention is presented. At power-up, execution begins in boot flash memory segment 0 (1125). The integrity of the main flash code space is checked and then control is passed to main flash memory segment 0 (1145). At this point, boot flash memory segments 0 (1125), 1 (1120), 2 (1115) and 3 (1110) are idle and main flash memory segments 0 (1145), 1 (1140), 2 (1135) and 3 (1130) are selected for program execution.

[0052] Turning now to FIG. 12, a block diagram that illustrates the transition that happens when code in main flash memory is preparing to transfer control to code in boot segment 0 in accordance with one embodiment of the present invention is presented. The transfer of program control is initiated when the embedded processor (reference numeral 425 of FIG. 4) receives a code update command.

[0053] Turning now to FIG. 13, a block diagram that illustrates the state of main flash memory and boot flash memory when passing program control to boot flash memory in accordance with one embodiment of the present invention is presented. After program control is passed to main flash memory segment 0 (1345), control is passed to boot flash memory segment 0 (1325).

[0054] Turning now to FIG. 14, a block diagram that illustrates the state of main flash memory and boot flash memory after passing program control to boot flash memory in accordance with one embodiment of the present invention is presented. After program control is passed to boot flash memory segment 0 (1425), main flash memory segments 0 (1445), 1 (1440), 2 (1435) and 3 (1430) are idle and code is run from boot flash memory segments 0 (1425), 1 (1420), 2 (1415) and 3 (1410).

[0055] Once a corrupt main flash memory segment has been detected and program control has been passed to boot flash memory, main flash code must be updated. Since code space is read-only, the main flash memory segments are mapped one at a time to

data space and then new code is written. This is illustrated with reference to FIG. 15, below.

[0056] Turning now to FIG. 15, a block diagram that illustrates the state of main flash memory and boot flash memory when mapping a main flash code segment to data space in accordance with one embodiment of the present invention is presented. While code is being executed in boot flash memory segments 0 (1514), 1 (1512), 2 (1510) and 3 (1508), the main flash memory segments are mapped one at a time to data space. Segment 1540 shows mapping main flash code segment 0 to data space. The same segment (1540) is also used for mapping other main flash code segments.

[0057] Figures 16-17 illustrate the state of code memory for a process that begins with detecting a corrupt boot flash memory segment (FIG. 16) and ends with transferring program control to main flash memory (FIG. 17).

[0058] Turning now to FIG. 16, a block diagram that illustrates the state of main flash memory and boot flash memory when a corrupt boot flash memory segment is detected in accordance with one embodiment of the present invention is presented. Upon power-up, boot flash memory segment 0 (1625) executes. Corrupt boot segments 1 (1620), 2 (1615) and 3 (1610) are detected, at which point program control is passed to main flash memory segment 0 (1645).

[0059] Turning now to FIG. 17, a block diagram that illustrates the state of main flash memory and boot flash memory after program control is transferred to main flash memory upon detecting a corrupt boot flash memory segment in accordance with one embodiment of the present invention is presented. After program control is passed to main flash memory segment 0 (1745), program execution proceeds from main flash memory segments 0 (1745), 1 (1740), 2 (1735) and 3 (1730).

Detecting Corrupt Software Code

[0060] According to embodiments of the present invention, predetermined values ("marker values") are written to predetermined memory locations (marker locations) in a program that is downloaded to an embedded device. The marker values and marker locations use a correlation rule known to the CPU (reference numeral 415 of FIG. 4) and the embedded device. The correlation rule establishes a relationship between two or more marker values. Once the program is downloaded to the embedded device, the embedded device applies the correlation rule to the marker locations. Corrupted memory (FIG. 18) is indicated when a marker value differs from an expected value.

[0061] According to one embodiment of the present invention, marker values are changed for each code version, but the correlation rule used to compare the marker values remains the same.

[0062] According to one embodiment of the present invention, the correlation rule comprises comparing the content of the first memory location or memory unit of a segment with the last memory location of the segment.

[0063] According to one embodiment of the present invention, the correlation rule comprises comparing the content of the first memory location of a segment with the last memory location of the segment and indicating the segment is corrupt if the content of the first memory location does not equal the complement of the second memory location. Those of ordinary skill in the art will recognize that other comparisons may be made.

[0064] According to another embodiment of the present invention, the correlation rule comprises comparing the content of a memory location in a first memory segment with the content of a memory location in a second memory segment and indicating a corrupt segment if the content of the first memory location does not equal the complement of the second memory location.

[0065] Figures 19-21B illustrate exemplary correlation rules in accordance with embodiments of the present invention. Figure 19 illustrates main flash memory markers. Figure 20 illustrates boot flash memory markers. Figure 21A illustrates marker correlation corresponding to FIG. 19. Figure 21B illustrates marker correlation corresponding to FIG. 20.

[0066] Turning now to FIG. 19, a block diagram that illustrates the placement of markers in main flash memory in accordance with one embodiment of the present invention is presented. Figure 19 includes main flash memory segments F0 (1920), F1 (1910), F2 (1904) and F3 (1900). Each main flash memory segment (1920, 1910, 1904, 1900) includes markers that check the internal consistency of the segment. The markers occupy the first and last memory location of each segment. The markers may occupy other locations in a segment as well. Thus, M4 (1902) is compared with M12 (1930), M3 (1906) is compared with M11 (1936), M2 (1914) is compared with M10 (1944) and M1 (1958) is compared with M9 (1956). Marker M1 (1958) is used instead M0 (1924) to compare against M9 (1956), illustrating a case where a memory location is not available for use as a marker, such as, by way of example, when the location forms part of a power-up vector. These parings are listed at reference numeral 2100 of FIG. 21A.

[0067] Still referring to FIG. 19, each main flash memory segment (1920, 1910, 1904, 1900) also includes markers used to detect when a flash memory update fails at a segment boundary. Each marker pair includes markers from adjacent segments. A marker in the first segment is compared with a marker in the last segment. Thus, M8 (1928) is compared with M13 (1932), M7 (1934) is compared with M14 (1940), M6 (1942) is compared with M15 (1952) and M5 (1954) is compared with M16 (1926). These parings are listed at reference numeral 2105 of FIG. 21A.

[0068] Turning now to FIG. 20, a block diagram that illustrates the placement of markers in boot flash memory in accordance with one embodiment of the present

invention is presented. Figure 20 includes boot flash memory segments B0 (2022), B1 (2016), B2 (2012) and B3 (2008). To check the internal consistency of each segment, M10 (2044) is compared with M24 (2006), M20 (2050) is compared with M2 (2014), M9 (2056) is compared with M23 (2018) and M19 (2062) is compared with M1 (2058). As illustrated in FIG. 20, marker M1 (2058) is not the first location in boot segment B0 (2022) because TBD. These parings are listed at reference numeral 2110 of FIG. 21B.

[0069] Still referring to FIG. 20, each boot flash memory segment (2022, 2016, 2012, 2008) also includes markers used to detect when a flash memory update fails at a segment boundary. M21 (2038) is compared with M5 (2054), M15 (2066) is compared with M18 (2048) and M22 (2064) is compared with M6 (2042). These parings are listed at reference numeral 2115 of FIG. 21B.

[0070] Figures 21A-21B are tables that illustrate correlation between markers in accordance with embodiments of the present invention. Figure 21A illustrates correlation between markers in a main flash memory as shown in FIG. 19. Figure 21B illustrates correlation between markers in a boot flash memory as shown in FIG. 20.

[0071] Turning now to FIG. 22, a flow diagram that illustrates a method for detecting corrupted code in accordance with one embodiment of the present invention is presented. At 2200, a correlation rule between storage units in a program are defined. The correlation rule defines which storage units are to be compared and what relationship each storage unit should have to the storage unit with which it is compared. An

exemplary correlation rule may specify that the first and last storage units of each memory segment are to be compared and that the value of each storage unit should be the complement of the storage unit with which it is compared. Those of ordinary skill in the art will recognize that other storage units and other relationships between storage units may be used.

[0072] Still referring to FIG. 22, at 2205 execution begins in boot flash memory. At 2210, a determination is made regarding whether main flash memory has been corrupted. If main flash memory has been corrupted, an indication is made at 2220 and execution proceeds from boot flash memory at 2235. If main flash memory has not been corrupted, the code space is re-mapped to main flash memory locations. At 2230, execution proceeds from main flash memory.

[0073] Turning now to FIG. 23, a flow diagram that illustrates a method for determining whether main flash memory has been corrupted in accordance with one embodiment of the present invention is presented. Figure 23 provides more detail for reference numeral 2210 of FIG. 22. At 2300, a first marker is read. At 2305, a second marker corresponding to the first marker is read. At 2310, the correlation rule is applied. At 2315, a determination is made regarding whether the result of applying the correlation rule matches the expected result. If the two results do not match, a corrupted segment is indicated at 2320. If the two results match, a determination regarding whether more markers remain is made at 2325. This process continues for all markers defined for the

program until a corrupted segment is found or until all markers for the program have been examined.

[0074] Embodiments of the present invention have a number of advantages. Using a correlation rule to compare particular locations in a program downloaded to an embedded device provides a relatively efficient mechanism for detecting corrupted code. Time-consuming read-and-compare operations are no longer required because the correlation rule is known by and is directly applied by the embedded device itself. The correlation rule also obviates the need for specialized knowledge about particular code versions. Mapping memory as described also enables a relatively reliable and efficient means for updating code.

[0075] Embodiments of the present invention have been illustrated using embedded devices that use flash memory. However, those of ordinary skill in the art will appreciate that the invention may be applied to non-embedded devices as well. Additionally, those of ordinary skill in the art will recognize that other types of memory may be used.

[0076] While embodiments and applications of this invention have been shown and described, it would be apparent to those skilled in the art having the benefit of this disclosure that many more modifications than mentioned above are possible without departing from the inventive concepts herein. The invention, therefore, is not to be restricted except in the spirit of the appended claims.